λ-Calculus

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Subtitle: “Everything is Really a Function”

The purposes of this lecture is to demonstrate how functions themselves can represent arbitrary data structures.

- Know the three constructs of λ-calculus.
- Know the competitive advantage of λ-calculus.
- Know how to use functions to represent integers.
  - Know how to write addition, multiplication, and exponentiation.
- Know how to use functions to represent arbitrary types.
  - booleans
  - pairs
  - lists

- Contains three kinds of things:
  1. Variables: x
  2. Functions: $\lambda x.x$
  3. Application: $(\lambda x.x)\ y$

  ... and nothing else!
- “We don’t need no stinking integers!”
- This language is Turing Complete.
- Used extensively in research. The “little white mouse” of computer science.
- We can implement this very easily in OCaml.
  $\lambda x.x$ = `fun x -> x`

The semantics of λ-calculus are very easy. Remember that M,N,P are expressions, V is a value.

Variables $\dfrac{}{x \Downarrow x}$

Functions $\dfrac{}{\lambda x.M \Downarrow \lambda x.M}$

Application $\dfrac{M \Downarrow \lambda x.P \quad N \Downarrow V}{M\ N \Downarrow [V/x]P}$

What can you tell about this language?

Here's another version of λ-calculus.

Variables $\dfrac{}{x \Downarrow x}$

Functions $\dfrac{}{\lambda x.M \Downarrow \lambda x.M}$

Application $\dfrac{M \Downarrow \lambda x.P}{M\ N \Downarrow [N/x]P}$

What is different about this version of the language?

We will add one "rule" to these: $\dfrac{M \Downarrow \lambda x.N \quad N \Downarrow V}{M \Downarrow \lambda x.V}$

This is known as *reduction*, as opposed to *evaluation*. Can you see why this rule is not valid?

- The Lambda Calculus doesn't have numbers.
- A number n can be thought of as a potential: someday we are going to do something n times.

```
f0 = fun f x -> x
f1 = fun f x -> f x
f2 = fun f x -> f (f x)
f3 = fun f x -> f (f (f x))
```

A numeral n takes a function and a value, and applies the function n times to the value.

```
# let fShow m = m ( (+) 1 ) 0;;
val fShow : ((int -> int) -> int -> 'a) -> 'a = <fun>
# fShow f3;;
- : int = 3
```


- To increment a Church Numeral...

```
# let fInc m f x = f (m f x);;
val fInc : (('a -> 'b) -> 'c -> 'a) -> ('a -> 'b) ->
# fShow (fInc f3);;
- : int = 4
```

- First, apply `f` m times to `x`.
- Next, apply `f` once more to the result.

- Similar reasoning can yield addition and multiplication.

```
# let fAdd m n f x = m f (n f x);;
val fAdd : ('a -> 'b -> 'c) ->
           ('a -> 'd -> 'b) -> 'a -> 'd -> 'c = <fun>
# fShow (fAdd f3 f3);;
- : int = 6
# let fMul m n f x = m (n f) x;;
val fMul : ('a -> 'b -> 'c) ->
           ('d -> 'a) -> 'd -> 'b -> 'c = <fun>
# fShow (fMul f3 f3);;
- : int = 9
```

- Subtraction is *much* more complex.

```
# let fAdd m n f x = m f (n f x);;
# fAdd f2 f3;;
→ f2 f (f3 f x)
→ f2 f ((fun g y -> g (g (g y))) f x)
→ f2 f (f (f (f x)))
→ (fun g y -> g (g y)) f (f (f (f x)))
→ (fun y -> f (f y)) (f (f (f x)))
→ f (f (f (f (f x))))
```



```
# let fMul m n f x = m (n f) x;;
# fMul f3 f3;;
→ f3 (f3 f) x
→ f3 ((fun g y -> g (g (g y))) f) x
→ f3 (fun y -> f (f (f y))) x
→ (fun h z -> h (h z)) (fun y -> f (f (f y))) x
→ (fun y -> f (f (f y))) ((fun y -> f (f (f y)))
                          ((fun y -> f (f (f y))) x))
→ (fun y -> f (f (f y))) ((fun y -> f (f (f y)))
                          (f (f (f x))))
→ (fun y -> f (f (f y))) (f (f (f (f (f (f x))))))
→ (f (f (f (f (f (f (f (f (f x))))))))
```


Suppose we have a type S, with constructors $S_1 \ldots S_n$. We can represent a term $t = S_i(t_1, t_2, \ldots, t_m)$ by

$$\overline{t} = \lambda x_1 \ldots x_n . x_i \, \overline{t_1} \cdots \overline{t_m}$$

The x_i represent functions that say what to do if the term t turns out to be an instance of the constructor S_i.

Let $S_1 =$`true`, and $S_0 =$`false`.
Then, a boolean is a value which takes two functions. The first function is what to do when the boolean is `true`, the second is what to do when the boolean is `false`.

```
# let fTrue a b = a;;
val fTrue : 'a -> 'b -> 'a = <fun>
# let fFalse a b = b;;
val fFalse : 'a -> 'b -> 'b = <fun>
# let fIf c t e = c t e;;
val fIf : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

- How would you define `fAnd` and `fOr` with this system?

A pair is a type with one constructor $S_0 =$`pair`. We know that S_0 takes two arguments.
Then, a pair is a value that takes one function. The function itself takes two arguments, and tells what to do with the two halves of the pair.

```
# let fPair a b = fun select -> select a b;;
val fPair : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fun
# let fPi1 p = p (fun a b -> a);;
val fPi1 : (('a -> 'b -> 'a) -> 'c) -> 'c = <fun>
# let fPi2 p = p (fun a b -> b);;
val fPi2 : (('a -> 'b -> 'b) -> 'c) -> 'c = <fun>
```


- You can combine these techniques to create arbitrary data structures.
- A list contains two constructors: `Cons` and `Nil`.
- The `Cons` constructor contains a pair as data.

```
# let fNil cons nil = nil;;
val fNil : 'a -> 'b -> 'b = <fun>
# let fCons a b cons nil = cons a b;;
val fCons : 'a -> 'b -> ((('a -> 'b -> 'c) -> 'c) ->
# let fIsNil lst = lst (fun x y -> fFalse) fTrue;;
val fIsNil : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) ->
```

You will need to tell OCaml to use recursive types for this... but even then the type system starts to get in the way.

1. (126) Write a function `church n` which returns the Church Numeral for n. (Use recursion.)
2. (127) Write the church function for exponentiation. (Hint: the solution is shorter than multiplication.) Show a sample run for 3^2.
3. (128) How would you represent a tree using functions? (There are many ways to do this.)

- Write a function `church n` which returns the Church Numeral for n. (Use recursion.)

```
# let rec church n f x =
    match n with
      0 -> x
    | _ -> f (church (n-1) f x);;
val church : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

- Write the church function for exponentiation. (Hint: the solution is shorter than multiplication.) Show a sample run for 3^2.

Solution:

```
# let fPow m n = n m ;;
# fPow f3 f2
→ f2 f3
→ (fun f x -> f (f x)) f3
→ fun x -> f3 (f3 x)
→ fun x -> (fun g y -> g (g (g y))) (f3 x)
→ fun x -> fun y -> (f3 x) ( (f3 x) ( (f3 x) y))
→ fun x y -> (f3 x) ( (f3 x) (x (x (x y))))
→ fun x y -> (f3 x) (x (x (x (x (x (x y))))))
→ fun x y -> (x (x (x (x (x (x (x (x (x y))))))))
```


- How would you represent a tree using functions? (There are many ways to do this.)

```
# let mkLeaf n node leaf = leaf n;;
val mkLeaf : 'a -> 'b -> ('a -> 'c) -> 'c = <fun>
# let mkNode a b node leaf = node a b;;
val mkNode : 'a -> 'b -> ('a -> 'b -> 'c) -> 'd -> 'c
```


- You can use λ-calculus to represent itself using these techniques. You already have everything you need to do it. You can see the details in Torben Æ. Mogensen's paper Efficient Self-Interpretations in lambda Calculus, in the Journal of Functional Programming v2 n3.
- More information about using functions to represent objects can be found in J. Steensgaard-Madsen's paper Typed representation of objects by functions, in TOPLAS v11 n1.